Chohyeon Kim

Assumption & Simplification

Used pre-trained YOLOv8 instead of training a custom model.
Although it can be designed for horizontal scaling, I ran it as a single container/instance for simplicity and time constraints.
No paid services (e.g., cloud hosting, load balancers) were used.
In-memory queue and result storage were used for simplicity.
Implemented and tested on a single machine.
Assumed reasonable image sizes (e.g., up to 5MB per image) for processing; hence, compression and streaming were not implemented.
While not implemented, using a queue structure would make features like retrying failed image processing jobs easier to support.

Approach to solving the problem.

Initially, I considered a simple synchronous server where the upload and process images occur in a single request cycle. However, there are some limitations with this approach:

Synchronous Backend Bottleneck: In synchronous workflow, each image upload would block the server until image processing finishes. If one image takes a long time to update and process (e.g., large file), it can delay other requests. During traffic peaks, the server may become unresponsive or reject requests due to overload

To tackle this challenge, I identified two primary bottlenecks in the system:
1. Image Uploading (I/O-bound)
2. Image Processing (CPU-bound)

To address the two bottlenecks I designed the system to follow a sequential, async flow:

1. Upload Handling: When a client uploads an image, the FastAPI server receives it. It is async function, it does not block other process while waiting for Network I/O
2. Quicker Response to Client: Upon receiving the image, the server enqueues the job and immediately returns a batch_id to the client. This allows the client to get a response quickly, without waiting for processing to complete.
3. Background Queue Processing: Images are placed into a JoinableQueue, which allows safe communication between worker processes. The workers continuously fetch jobs from this queue and process them in parallel using multiprocessing.
4. Polling for Results: Since processing is asynchronous, the client must poll the /result/{job_id} endpoint to check whether the result is ready. This decouples the processing from the user experience.

Scalability strategy.

The system is designed to scale efficiently both across and within servers to handle high traffic.

1. Scaling Efficiency per Server: The FastAPI server handles requests asynchronously and enqueues them for background processing. Image processing tasks are pulled from the queue and run in parallel.
2. Horizontal Scaling: The system can be containerized using Docker and deployed across multiple instances. A load balancer can distribute incoming requests across these instances.
3. Regional Segmentation : If deployed globally, servers can be grouped by DNS-based routing to direct clients to the nearest region. This minimizes latency and distributes the global load more evenly.

Challenges Faced and Potential Improvements

1. Async Overhead for Low-Traffic Systems
In low-traffic environments, the overhead of setting up asynchronous I/O may result in slower performance compared to simpler synchronous approaches.

2. Queue - Memory Overload

1. Persistent Storage: Instead of queue, storing image data temporarily on disk (e.g., using file system or database) can reduce memory pressure but will increase I/O latency.

2. Request Limiting and Upload Guidelines: To Prevent overload at the entry point. Enforce limits on request size, number of concurrent uploads per client, and image resolution. This can reduce memory pressure. Encouraging or enforcing image compression before upload is another practical optimization.

3. Request with 100 Images - Streaming
Instead of loading entire image files into memory, streaming allows chunk-based reading and processing, which can reduce peak memory usage during I/O. This is more effective when working with very large files

4. Priority Queue for Request Handling
Since image processing is a long-running and resource-intensive task, performance gains depend on available system resources. In this setup, without resource isolation, high-priority requests (e.g., uploads) may suffer delays. To mitigate this, we can incorporate a priority queue mechanism to ensure time-sensitive tasks are handled first, improving responsiveness even under limited resources.

5. In-memory result storage

1. The system stores results in memory for clients to retrieve via `/result/{job_id}`. Results accumulate in memory. Use a scheduled background cleanup task or manually purge it after the job completes
2. In-memory result storage is not scalable, as each container maintains its own isolated memory. This leads to inconsistencies when scaling horizontally, and can be improved by refactoring to use a shared storage to enable consistent access and true horizontal scalability.

6. Polling Overhead
If clients repeatedly poll the `/result/{job_id}` endpoint. The backend may experience unnecessary load, especially during peak traffic.

1. Tune polling interval to a sensible default (e.g., every 2–3 seconds).
2. Set a result expiration window (e.g., 5–10 minutes). If the user does not retrieve the result within that time, it expires. Inform the user to retry the upload
3. Explore event-driven alternatives (notification system) for more efficient real-time result delivery when scaling further.

7. Monolithic Processing Design

In the current design, both image upload and processing are handled within the same backend service. This tightly coupled structure may limit flexibility and scalability at scale. Because any change or failure in one part can affect the entire service. It also makes it harder to independently scale components - for example, scaling up the processing logic would require duplicating the entire server leading to inefficient resource usage.

1. Decouple services: Separate the upload service from the processing service
    - Upload images to external storage (e.g., S3 or blob storage), and trigger processing asynchronously by providing just the image URL to the image processing service.
    - This way is easier to scale up specific parts (e.g., only processing layer) depending on where the bottleneck is. (Horizontal Scaling)
2. Batch-Level Parallelism: When a client uploads multiple images at once, the batch can be split into smaller sets and processed in parallel across multiple image processing servers and send result back to the user

LLM Usage

I used ChatGPT to support the writing of the README, report, and code comments. All core ideas, architectural design decisions, and implementation logic were developed by me. The LLM was mainly used to validate design approaches and help with tooling (e.g., FastAPI setup, YOLOv8 integration- food_dectector.py) and dockerfile - docker-compose.yaml and test script for client side.